AutoPilot: Message Passing Parallel Programming for a Cache Incoherent Embedded Manycore Processor

Ben Kelly
Google Canada
151 Charles St. W. Suite 200
Kitchener, ON, Canada N2G 1H6
+1-519-513-5604
btk@google.com

William B. Gardner
School of Computer Science
University of Guelph
Guelph, ON, Canada N1G 2W1
+1-519-824-4120 x52696
wgardner@socs.uoguelph.ca

Shorin Kyo
Renesas Electronics
1763, Shimonumabe, Makahara-ku
Kawasaki, Kanagawa
211-8668, Japan
+81-44-435-5446
shorin.kyo.wz@renesas.com

ABSTRACT

The Renesas Electronics IMAPCAR2 embedded realtime image processor combines a single core with a 128-way SIMD array. At runtime, sections of the SIMD array can be reconfigured as additional CPU cores, interconnected via a message ring. Effective use is made difficult by the low-level message passing API and lack of cache coherency between processors. The AutoPilot library addresses this by providing a high-level message-oriented parallel programming model mirroring that of Pilot, itself a wrapper around the Message Passing Interface (MPI) for cluster computing. AutoPilot shows that Pilot's processes-and-channels architecture is a viable choice for parallel programming on cache-incoherent multicore and manycore architectures. It provides a simpler API for programmers, with built-in safety checks that eliminate some common sources of errors. Since the IMAPCAR2 is targeted chiefly at automotive applications, open source AutoPilot has a large degree of MISRA-C compliance.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent programming—*parallel programming*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*software libraries*; D.4.1 [**Operating Systems**]: Process Management—*deadlocks*; D.4.4 [**Operating Systems**]: Communications Management—*message sending*

General Terms

Performance, Design, Languages

Keywords

Parallel programming; embedded; multicore; manycore; Pilot; cache coherence; message passing; IMAPCAR2; XC core

1. INTRODUCTION

Embedded multicore and manycore processors may feature idiosyncratic architectures that present special challenges for their software programmers. Far from being *intentionally* difficult to program, these architectures are a natural by-product of normal embedded design trade-offs intended to maximize certain capabilities and performance characteristics, while minimizing other criteria such as area and power.

Three examples of challenging features and their consequences for software are lack of cache coherency, complex communication mechanisms, and subtle failure modes. Dispensing with cache coherency leaves programmers with the appearance of global shared memory, but where global variables do not behave as they expect. The "natural" programming model of pthreads becomes treacherous and laden with pitfalls for the unwary. The programmer may have numerous choices to make among a variety of low-level communication APIs, so that even writing a simple parallel program can entail a lengthy learning curve. Unwittingly violating requirements on buffer sizes, address alignments, and network capacities may result in full or partial communication failures, perhaps without meaningful error feedback, and even deadlocks. Such problems may be extremely difficult to diagnose.

In this work, we insert a thin communication library layer on top of the vendor-provided API in order to (1) direct the parallel programmer away from global memory by offering a simple message passing API, (2) provide a small set of abstractions useful for structuring a parallel solution, and (3) silently furnish a selection of working techniques for runtime processor reconfiguration and communication, while hiding low-level details and giving runtime diagnosis of programmer usage errors. This approach goes far toward overcoming the three challenges outlined above.

MES '13, June 23 - 24 2013, Tel-Aviv, Israel

2. BACKGROUND

The IMAPCAR2 [3], whose microarchitecture is called the *eXtensible Computing Core* or XC Core, was developed for high-performance embedded image processing applications especially in the automotive world. It consists of a RISC central processor (CP) and an array of RISC SIMD Processing Elements (PEs) each with its own on-chip image memory (IMEM), connected in a ring network for fast communication between adjacent PEs.

Dynamic reconfiguration capability allows software to select, at runtime, a trade-off between SIMD and MIMD operation (called *MP mode*). A model containing 128 SIMD PEs can be reconfigured to a cluster-on-chip of 32 independently-executing processing units (PUs) plus the CP. There is also a *mixed mode* where half the PEs remain for SIMD use by the CP while half reconfigure as PUs.

The PUs are interconnected with two networks, the *C-ring* allowing them to pass 16-byte messages directly between PUs (including the CP), and the *M-ring* connecting the PUs to the DMA hardware used for external memory access.

The memory model can essentially be divided into four types: *program memory* (PMEM), *data memory* (DMEM), *external memory* (EMEM), and *internal* or *image memory* (IMEM). In MP mode, IMEM is used for the data and instruction caches on the PUs, but there are multiple DMEM areas: each PU is assigned a memory region when it starts up, which it uses for stack and heap.

This picture is further complicated by the lack of cache coherency hardware—there is no built-in support for keeping the caches attached to different processors in sync. If two processors have the same variable in cache, and one updates it, the other will not see that update, but will continue to use its old value; if both update it,

they will each see their own update, and it is unspecified which update "wins" when their cache contents are copied back to main memory—which will not happen automatically until cache pressure requires the replacement of that cache line. While the caches can be bypassed entirely, this is not recommended, as cache access is many times faster than access to external memory, and is not subject to bottlenecks as when multiple processors are accessing the same external memory. Using shared memory requires extreme caution; it is preferred to avoid it in favour of message passing.

2.1 IMAPCAR2 Programming Issues

IMAPCAR2 programming uses *1-Dimensional C* (1DC), a variant of the C programming language with extensions for convenient use of SIMD capabilities, mostly irrelevant to AutoPilot.

The 1DC Standard Library, lib1dc, serves as both the C standard library and system library for IMAPCAR2 programs. Its functions for accessing specialized features of the chip are classified into memory functions, operation functions, and MP mode functions, the latter for managing PUs and sending messages.

Three major issues with parallel programming using the standard lib1dc library are described below.

MP Mode Configuration and Management. Activating one or more PUs involves providing each with a DMEM region and process entry point, and then starting them. The CP can block until all PUs finish execution. There is a PU suspend/resume feature, but this is not exposed by the AutoPilot API.

Inter-PU Communication. The primary means of interprocess communication (IPC) is the message, a data packet sent using the C-ring. The IMAPCAR2 message interface comprises three types of message (standard, synchronous, and interrupt), plus some convenience functions for sending bulk data that build on those.

Messages are sent and received in a multistep process involving filling in a header structure with message type, message ID and broadcast flag. The payload, if any, is assigned to struct members. The send function blocks until the message is successfully placed on the C-ring. The receiver follows the same steps, with the receive function blocking until an appropriate message arrives.

A destination is assigned by message ID and/or PU ID. Given multiple valid receivers, whichever PU sees the message first on the C-ring will receive it—the other receivers remain blocked.

The message-passing API, while initially daunting in its multiplicity and complexity of options, is not that hard to use, but it is very easy to *mis*use. Sending a message with the wrong ID or to the wrong PU will generate no warnings, it will simply result in a message on the C-ring that will never be received—or be received by the wrong process. Similarly, a block size mismatch between the sender and receiver will result in undefined behaviour.

A broadcast messages has no guarantee of reaching *N unique* PUs; rather, it will be received *N* times before it is removed from the C-ring. A single broadcast message can be received multiple times by the same PU which can result in unexpected and problematic behaviour. Worse, since block send uses broadcast messages internally, if there are multiple PUs waiting to receive a block message with the same ID, it will end up split between them!

There is no intrinsic support for message ordering. Messages sent sequentially, even from the same PU, may be received in a different order than they were sent. Finally, there is the possibility of *C-ring flooding*. The C-ring has a limited number of slots, and once that many messages are in flight, new senders will be blocked until one or more circulating messages is received. This, makes it extremely easy to deadlock everyone, but the programmer may be unaware of this limitation, or fail to manage it properly.

Global Memory Safety. Two ways of coping with lack of cache coherency are manual cache management (invalidation and forcing write-back) and bypassing the cache entirely. However, invalidating a cache entry affects all addresses in the same 512-byte cache line, and forcing the write-back of some variable writes the entire cache line containing it. The result is that a PU can inadvertently change the values in other PUs' global variables which happen to share the cache line it is trying to update, creating difficult-to-diagnose sources of inter-thread interference.

Alternatively, reads or writes to a variable declared as `outside` will bypass the cache and access main memory directly, but at a cost of being 3-4x slower than access to cached variables.

It is exactly because of these treacherous conditions surrounding the use of global memory, for which the ordinary shared-memory multicore programmer is simply unprepared, that message passing is to be preferred on this architecture. This also removes the need for mutex protection of globals, which have their own opportunities for misuse leading to data corruption or deadlocks. With Pilot-style message passing, each process can still make fast access to its cache, but manual cache management is avoided.

2.2 Parallel Programming with Pilot

Pilot [1] is a cluster computing library created at the University of Guelph. It is a wrapper around the Message Passing Interface (MPI) commonly used in cluster computing, with a much smaller and simpler API (under two dozen functions vs. MPI's hundreds) and a number of helpful features such as automatic deadlock detection. It is "a friendly face for MPI," which can be deployed on any MPI-based platform to provide a set of communication functions with an API strongly inspired by C's printf and scanf.

Pilot is conceptually based on C.A.R. Hoare's Communicating Sequential Processes formalism (CSP), structured around *processes* and *channels*. A process is a single independent thread of execution; a channel, a dedicated, unidirectional line of communication between two processes. Channels are untyped; the type of each message is specified when read or write is called, thus one channel can be used to send differently typed messages. The API uses a format string to specify the types and lengths of data, and subsequent arguments with the values to be sent or locations to fill.

Channels are the only means for IPC, a restriction that aids in program analysis and permits debugging tools such as Pilot's deadlock detector. Since Pilot enforces IPC integrity via channels, one category of parallel program logic and coding errors is avoided.

Pilot also supports *collective communication*, read or write operations involving multiple channels. These are performed on *bundles*, collections of channels with a common endpoint. For example, *broadcast* writes the same message to every channel in a bundle. *Select* identifies an individual channel in a bundle which is ready for reading, so that a message can be read from it without blocking. Select is an efficient way for a master process to monitor completion of workers who have different quantities of work to do.

3. AUTOPILOT API

Mapping the Pilot approach onto the IMAPCAR2 involves these ingredients (more implementation details in Section 4):

Processes: Each PU is available to run a process defined by some C function. Processes can run different functions, or multiple instances of the same function, i.e, the same style of thread initiation as pthread_create, with the difference that PUs cannot be "oversubscribed" by multiple threads. The process-per-PU approach is compatible with scalable parallel program organization and eliminates thread scheduling overhead. The CP will continue to run the code in the main() function, and may serve as the master in a master/worker pattern, if desired.

Channels: The programmer creates channels between any pair of processes (PU or CP). For collective operations, bundles are created in which the common endpoint can be the CP or any PU.

When the programmer wishes to switch into MIMD or mixed mode processing, the AutoPilot *configuration phase* is initiated by calling PI_Configure. Next, functions are called to create process and channel definitions. Finally, calling PI_StartAll will change the chip into MP or mixed mode, and each configured PU will start running its respective process function, while the CP continues executing statements following PI_StartAll. This is called the *execution phase*, during which the various processes will carry out communication by calling PI_Write, PI_Read, and collective functions. If mixed mode is in effect, the CP may continue using 1DC operations that invoke SIMD instructions.

MIMD processing is brought to an end when all of the PU processes have returned from their respective functions, and the CP has called PI_StopMain. At that point, the chip will revert to SIMD mode and the CP will continue executing statements following PI_StopMain. The above cycle of phases may be repeated.

This provides the programmer with solutions to the issues identified in Section 2.1: MP mode configuration is accomplished via PI_Configure and PI_StartAll, which invoke the necessary SDK functions to allocate memory, create stack and heap storage for each process, switch the processor's mode, and start execution at each designated process function with the given arguments. Inter-PU communication is handled transparently by PI_Write and PI_Read as they look up the PU destination corresponding to the designated channel and carry out a handshaking protocol designed to avoid C-ring flooding. Global memory safety is realized as a by-product of using AutoPilot messages for IPC instead of risky global variables.

Two snippets of samples code are provided to compare using the AutoPilot API versus the native SDK. Figure 1 illustrates switching into MIMD mode with *N* PUs, each using a region of DMEM `dmem_size` bytes, starting from address `dmem_area`. PI_CreateProcess makes each PU run an instance of work_func(pu). PI_CreateChannel defines a channel from the CP (known symbolically as PI_MAIN) to each worker process, for subsequent use in dispatching data. PI_StartAll triggers all the PUs to start their work functions, and PI_StopMain blocks until they all exit. Here, the AutoPilot API roughly matches the native SDK, but provides more automation.

Figure 2 shows the CP dispatching data to the workers via PI_Write, and each worker reading its channel via PI_Read. Note the syntax mirroring C's fprint and fscanf: the "%d" format conveniently specifies a single decimal value. (A full range of format type codes are available for scalars, and for fixed-length and variable-length arrays.) The native SDK requires additional function calls to set up message headers, and the programmer would likely need to consult documentation regarding arguments and message fields. In this simple case the message ID can be left as 0.

Programmers can control the extent of run-time error checking by the AutoPilot library and the amount of diagnostic feedback—reporting even the exact source file and line that caused the error—by setting the preprocessor symbol PI_DEBUG.

MISRA-C is a standard for the use of the C programming language in automotive and other critical applications expressed in 142 rules. Beyond its unavoidable use of 1DC, AutoPilot impinges on these rules chiefly by utilizing variable arguments lists (varargs) for messaging functions, and macros needed to implement call site tracing. Details can be found in Appendix B of [2].

```
PI_CHANNEL toWorkers[N];
PI_Configure(N, dmem_size, dmem_area);

for (pu = 0; pu < puno ; ++ pu) {
  PI_PROCESS p=
     PI_CreateProcess(work_func, pu, NULL);
  toWorkers[pu]=PI_CreateChannel(PI_MAIN, p);
}
PI_StartAll();
/* CP computation omitted */
PI_StopMain();

//-----Native SDK version-----//
if (N > PUNO/2) Ix_start_mode ( IX_MP_MODE );
else Ix_start_mode ( IX_MIXED_MODE );

for (pu = 0; pu < puno ; ++ pu) {
  Ix_start_pu(pu, work_func, dmem_area,
              dmem_size, NULL);
  dmem_area += dmem_size;
}
/* CP computation omitted */
Ix_join(0, 0);
Ix_start_mode( IX_SIMD_MODE );
```

Figure 1. Initialization code

```
  /* write data to each worker */
for (pu = 0; pu < N; ++pu) {
  PI_Write(toWorkers[pu], "%d", data);
}
/* worker reads its channel */
PI_Read(toWorkers[pu], "%d", &mydata);

//-----Native SDK version-----//
  /* write data to each worker */
ix_MSG msg;
for (pu = 0; pu < puno ; ++ pu) {
  msg.hd = ix_msg_header (pu , 0, 0);
  msg.d0 = data;
  ix_send_msg (& msg , 0);
}
/* worker reads its message */
ix_MSG msg;
msg.hd = ix_msg_header(ix_cp_id(), 0, 0);
ix_recv_msg(&msg, 0);
mydata = msg.d0;
```

Figure 2. Write/read code

4. IMPLEMENTATION

This section describes how the library implements AutoPilot's API (Section 3) while solving the issues identified in Section 2.1.

Mode switching and process startup. Mode switching is initiated when the CP calls PI_Configure with the desired *count* of active PUs and a *dmemsize* indicating how much DMEM each should have available. If an optional *dmemptr* is provided, it is partitioned into a DMEM region for each PU obeying the 1KB alignment requirement; if not, AutoPilot allocates memory for each PU. PUs are initialized with a small function that initially does nothing but wait for configuration information to be supplied via PI_CreateProcess.

When the CP calls PI_StartAll, AutoPilot first performs a cache write-back to ensure that all PUs have accurate values for bundle and channel tables, and then sends a broadcast message to all PUs. The ones that have not been configured will react by halting themselves, and the others will call their process functions, halting only after this function returns.

In reverting to SIMD mode, PI_StopMain blocks until all of the PUs have finished executing. It frees any DMEM that it allocated earlier, and finally switches modes whereupon the CP will continue executing statements following PI_StopMain.

Channel writing and reading. A number of solutions for avoiding C-ring flooding were considered, including polling ready-to-read, ready-to-write flags in global memory, adopting a central "manager process" to maintain channel status and give permission

to communicate, and letting the CP serve as the communication manager via interrupt service functions. The current adopted solution is a refinement of the interrupt-based approach. When PI_Write is called, the writer sends an interrupt message to the reader, immediately followed by a "ready to send" (RTS) message containing information about the attempted write. The interrupt handler on the reader receives the RTS message, guaranteeing that it doesn't remain on the C-ring indefinitely, and sets the channel's *has_data* flag. The writer then waits for a "ready to receive" (RTR) message from the reader, which contains a pointer to the results of parsing the format string passed to PI_Read. The writer compares the two format strings to ensure that there are no type mismatches before sending the data, a means for detecting erroneous use by application code and exposing bugs.

After handshaking and format verification, the writer sends a single C-ring message for each data argument passed to PI_Write. In cases where the format string contains multiple codes (such as "%d %d"), each one is sent in its own message. Each message carries a sequence number which allows PI_Read to store message data into the proper variable or array argument.

In cases where a data item is an array of values—for example, the format "%32u"—the data itself is not sent in the C-ring message. As C-ring messages have a limit of 12 bytes, large data must be either split across multiple messages or sent in a manner that bypasses the C-ring. PI_Write ensures that the contents of the array are written back to main memory, then sends a pointer to the array to the reader, which copies the data into its destination buffer.

Bundle broadcasting. Broadcasting is encumbered by C-ring flood avoidance. The efficient hardware broadcast feature is not employed until each receiving process returns RTR, after which the writer broadcasts the actual data. At this point efficiency gains over PI_Write are realized: with all of the readers waiting for a message with the same tag, the sender can send a single broadcast message, which will then be received by all of the readers. Concurrent collective operations do not interfere because they use different message IDs.

5. RESULTS

In terms of code volume, the short samples in Section 3 convey the general trend: Except for configuration—where AutoPilot requires additional calls to create channels and bundles—AutoPilot code needs fewer function calls, and its API has fewer and simpler arguments, while the programmer benefits from the intuitive similarity to C's fprintf and fscanf. Furthermore, AutoPilot provides more automation "under the hood" and ensures freedom from C-ring flood-induced deadlocks.

In terms of memory footprint, AutoPilot programs link with lib1dc (1DC compiler's runtime), but do not need the pthreads library. Compared to lib1dc, pthreads adds another 12% of program code, while AutoPilot is smaller at 8%. However, there is a large contrast in DMEM footprint: AutoPilot uses almost 9KB for bookkeeping, but pthreads requires 40KB.

Timing tests were carried out on mode switching, simple messages, broadcasts, and array messages. Except for mode switching, this prototype version of AutoPilot lags behind the native SDK.

Mode switching time. The time to start *N* PUs in AutoPilot is linear in *N*, rising to 280 µs for 32 PUs, compared to 180 µs for ix_start_pu. AutoPilot needs extra time to send work function arguments via C-ring messages.

Small message time. In this benchmark, a small message—a single integer fitting into the payload of ix_send_msg—is sent from the CP to one or more PUs. The measured time for *N* PUs includes launching *N* processes, the CP calling PI_Write in a loop, and each process calling PI_Read once and exiting. Timing stops when the CP joins the terminated PUs. Regardless of *N* (1-32), AutoPilot was about 10 times slower than the native SDK. Profiling showed that the sender spends most of its time (72%) just waiting for the receiver to reply to its RTS message. The receiver, meanwhile, spends considerable time (56%) waiting for the RTS message, but also takes a long time to reply due to the cache write-back required to share the results of its format string parsing with the sender. The maximum time for AutoPilot to send to 32 PUs was about 1 ms, which is considered excessive for image processing applications that must handle a frame every 33 ms.

Broadcast time. The same pattern as for small messages—a 10x slowdown—applies to broadcasts because the same handshake is in use. The maximum AutoPilot time was still around 1 ms, just a little faster than sending 32 individual messages, which illustrates its current underutilization of the hardware broadcast mechanism.

Large message time. AutoPilot sends these messages by copying them into EMEM and then sending only a pointer via the C-ring, from which the receiver copies into the destination buffer. The native way is to use ix_send_msg_blk to pack the entire buffer into a sequence of eight-byte messages, along with sequence information for out-of-order arrival, sending them all via the C-ring. Tests were run with power-of-2 size buffers ranging from 4 to 2048 bytes. After deducting for the fixed overhead of format string parsing and handshaking, it was apparent that the pointer-passing strategy was inferior to the all-messages approach.

6. CONCLUSION AND FUTURE WORK

AutoPilot was created to provide a simpler, safer API for the IMAPCAR2. The Pilot API provides a single, consistent interface for managing process creation and communicating between processes, following printf-style conventions that are immediately familiar to any C programmer. The application programmer no longer needs to worry about choosing between several options for the same operation, and runtime verification of channel endpoints and format strings eliminates the possibility of sending messages to the wrong process or sending the wrong data types. Automatic handshaking between processes eliminates the possibility of C-ring floods, and call site tracing makes it much easier to debug errors in application code when they do occur. The core features of Pilot are implemented on the IMAPCAR2. The Pilot API is demonstrated to be usable on this architecture with only minor alterations, and AutoPilot is also quite compact, making it an easy addition to any program.

The current conservative strategy for avoiding C-ring flooding turned out to be glaringly inefficient. It can be replaced by another method of ensuring that a reader is waiting before a writer's messages are dispatched, for example, a circulating status message containing read-ready and write-ready flags for each channel. Other future work will implement more collective operations and provide a deadlock checker like Pilot's.

7. REFERENCES

[1] J. Carter, W.B. Gardner, and G. Grewal. The Pilot approach to cluster programming in C. In *Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-10), Proc. of the 24th IEEE International Parallel & Distributed Processing Symposium, Workshops and Phd Forum*, pages 1–8, April 2010.

[2] Benjamin Kelly. AutoPilot: A message-passing parallel programming library for the IMAPCAR2. Master's thesis, School of Computer Science, University of Guelph, Guelph, Ontario, Canada, Feb. 2013.

[3] Shorin Kyo, Takuya Koga, Lieske Hanno, Shouhei Nomoto, and Shin'ichiro Okazaki. IMAPCAR2: A dynamic SIMD/MIMD mode switching processor for embedded systems. In *Hot Chips 21: Symposium on High-Performance Chips*, Stanford University, Stanford, California, August 2009.